Boost.TR1

Copyright © 2005 John Maddock

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Introduction

The TR1 library provides an implementation of the C++ Technical Report on Standard Library Extensions. This library does not itself implement the TR1 components, rather it's a thin wrapper that will include your standard library's TR1 implementation (if it has one), otherwise it will include the Boost Library equivalents, and import them into namespace `std::tr1`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Usage

There are two things you need to decide before using the Boost.TR1 library: whether to use your standard library's native TR1 implementation (if it has one), and which include style to use.

Whether to use Your Native TR1 Library

If your standard library implements the TR1, and you want to make use of it, rather than use the Boost equivalents, then you will need to take some explicit action to enable it: this may be a pre-processor define, a special compiler switch, or a different include path. You will need to consult your compilers documentation to find out which of these actions you need to take.

Provided Boost is correctly configured, everything should now "just work", and code written to use Boost.TR1 will include your standard library's native headers rather than the Boost ones.

Header Include Style

There are two ways you can include the Boost.TR1 headers, for example if you are interested in shared_ptr then you can either use:

```
#include <boost/tr1/memory.hpp>
```

or:

```
#include <memory>
```

The first option is the preferred method for other Boost libraries to use. The second option is standard-conforming, but requires that you add `boost-install-path/boost/tr1/tr1` to your compiler's include search path. Note that you must not copy the headers in boost/tr1/tr1 into a directory called "include", doing so will cause them to cease working.

Important Note #1
The include path order is very important if you want this library to work correctly. If you get compiler errors then suspect the include paths. The correct order is:
1) boost-root/boost/tr1/tr1
2) boost-root
3) Any other standard library replacements (STLport for example).
4) Your regular standard library.

Important Note #2: Borland C++ Users
Borland's compiler has a particularly broken form of `#include`, that will actually look for a file named `array.h` if you `#include <array>`. In order to make this library work with Borland's compiler you will need to set up the include paths as follows:
1) boost-root/boost/tr1/tr1/bcc32
2) boost-root/boost/tr1/tr1
3) boost-root
4) Any other standard library replacements (STLport for example).
5) Your regular standard library.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Important Note #3: Sun C++ Users
Sun's compiler has a particularly interesting form of `#include`, that will actually look for a file named `array.SUNWCCh` if you `#include <array>`. In order to make this library work with Sun's compiler you will need to set up the include paths as follows:
1) boost-root/boost/tr1/tr1/sun
2) boost-root/boost/tr1/tr1
3) boost-root
4) Any other standard library replacements (STLport for example).
5) Your regular standard library.

Configuration

Configuring Boost.TR1 is no different to configuring any other part of Boost; in the majority of cases you shouldn't actually need to do anything at all. However, because Boost.TR1 will inject Boost components into namespace std::tr1 it is more than usually sensitive to an incorrect configuration.

The intention is that Boost.Config will automaticaly define the configuration macros used by this library, so that if your standard library is set up to support TR1 (note that few are at present) then this will be detected and Boost.TR1 will use your standard library versions of these components rather than the Boost ones.

If you would prefer to use the Boost versions of the TR1 conponents rather than your standard library, then either: include the Boost headers directly

```
#include <boost/regex.hpp>

boost::regex e("myregex"); //etc
```

Or else don't enable TR1 in your standard library: since TR1 is not part of the current standard, there should be some option to disable it in your compiler or standard library.

The configuration macros used by each TR1 component are documented in each library section (and all together in the Boost.Config documentation), but defining BOOST_HAS_TR1 will turn on native TR1 support for everything (if your standard library has it), which can act as a convenient shortcut.

Note for gcc users
Boost.TR1 does not currently enable gcc's native TR1 implementation as this is currently in an early stage of development. However, you may choose to do so by defining BOOST_HAS_GCC_TR1.

TR1 By Subject

Reference Wrappers.

```
#include <boost/tr1/functional.hpp>
```

or

```
#include <functional>
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The Ref library is a small library that is useful for passing references to function templates (algorithms) that would usually take copies of their arguments. It defines the class template `reference_wrapper<T>`, and the two functions `ref` and `cref` that return instances of `reference_wrapper<T>`. Refer to Boost.Bind for more information.

```
namespace std {
namespace tr1 {

template <class T> class reference_wrapper;

template <class T> reference_wrapper<T> ref(T&);
template <class T> reference_wrapper<const T> cref(const T&);
template <class T> reference_wrapper<T> ref(reference_wrapper<T>);
template <class T> reference_wrapper<const T> cref(reference_wrapper<T>);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_REFERENCE_WRAPPER if your standard library implements this part of TR1.

Standard Conformity: The Boost version of this this component does not currently support function call invocation (2.1.2.4), or derivation from std::unary_function or std::binary_function (2.1.2 paragraphs 3 and 4).

The Boost version is not implicitly convertible to T& as the TR requires.

Smart Pointers.

```
#include <boost/tr1/memory.hpp>
```

or

```
#include <memory>
```

The `shared_ptr` class template stores a pointer to a dynamically allocated object, typically with a C++ new-expression. The object pointed to is guaranteed to be deleted when the last `shared_ptr` pointing to it is destroyed or reset. For more information refer to the shared_ptr and weak_ptr documentation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

class bad_weak_ptr;

// [2.2.3] Class template shared_ptr
template<class T> class shared_ptr;

// [2.2.3.6] shared_ptr comparisons
template<class T, class U> bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U> bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U> bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);

// [2.2.3.8] shared_ptr specialized algorithms
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

// [2.2.3.9] shared_ptr casts
template<class T, class U> shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);
template<class T, class U> shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);

// [2.2.3.7] shared_ptr I/O
template<class E, class T, class Y>
basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

// [2.2.3.10] shared_ptr get_deleter
template<class D, class T> D * get_deleter(shared_ptr<T> const& p);

// [2.2.4] Class template weak_ptr
template<class T> class weak_ptr;

// [2.2.4.6] weak_ptr comparison
template<class T, class U> bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

// [2.2.4.7] weak_ptr specialized algorithms
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);

// [2.2.5] Class enable_shared_from_this
template<class T> class enable_shared_from_this;

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_SHARED_PTR if your standard library implements this part of TR1.

Standard Conformity: There are no known deviations from the standard when using the Boost version of this component.

Class template result_of.

```
#include <boost/tr1/functional.hpp>
```

or

```
#include <functional>
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The class template `result_of` helps determine the type of a call expression. Given an lvalue `f` of type `F` and lvalues `t1`, `t2`, `...`, `tN` of types `T1`, `T2`, `...`, `TN`, respectively, the type `result_of<F(T1, T2, ..., TN)>::type` defines the result type of the expression `f(t1, t2, ...,tN)`. The implementation permits the type `F` to be a function pointer, function reference, member function pointer, or class type. For more information refer to the Boost.Utility documentation.

```
namespace std {
namespace tr1 {

template <class T>
struct result_of
{
   typedef unspecified type;
};

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_RESULT_OF if your standard library implements this part of TR1.

Standard Conformity: No known problems.

Function template mem_fn.

```
#include <boost/tr1/functional.hpp>
```

or

```
#include <functional>
```

`std::tr1::mem_fn` is a generalization of the standard functions `std::mem_fun` and `std::mem_fun_ref`. It supports member function pointers with more than one argument, and the returned function object can take a pointer, a reference, or a smart pointer to an object instance as its first argument. `mem_fn` also supports pointers to data members by treating them as functions taking no arguments and returning a (const) reference to the member. For more information refer to the Boost.Mem_fn documentation.

```
namespace std {
namespace tr1 {

template <class R, class T> unspecified mem_fn(R T::* pm);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_MEM_FN if your standard library implements this part of TR1.

Standard Conformity: The Boost implementation does not produce functors that inherit from `std::unary_function` or `std::binary_function`, nor does it function correctly with pointers to volatile member functions (these should be extremely rare in practice however).

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Function Object Binders.

```
#include <boost/tr1/functional.hpp>
```

or

```
#include <functional>
```

`std::tr1::bind` is a generalization of the standard functions `std::bind1st` and `std::bind2nd`. It supports arbitrary function objects, functions, function pointers, and member function pointers, and is able to bind any argument to a specific value or route input arguments into arbitrary positions. `bind` does not place any requirements on the function object; in particular, it does not need the `result_type`, `first_argument_type` and `second_argument_type` standard typedefs. For more information refer to the Boost.Bind documentation.

```
namespace std {
namespace tr1 {

// [3.6] Function object binders
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;
template<class F, class T1, ..., class Tn > unspecified bind(F f, T1 t1, ..., Tn tn );
template<class R, class F, class T1, ..., class Tn > unspecified bind(F f, T1 t1, ..., Tn tn );

namespace placeholders {
   // M is the implementation-defined number of placeholders
   extern unspecified _1;
   extern unspecified _2;
   .
   .
   .
   extern unspecified _M;
}

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_BIND if your standard library implements this part of TR1.

Standard Conformity: The traits classes `is_placeholder` and `is_bind_expression` are not supported by the Boost implementation.

The named return value syntax isn't supported if the object being bound is a function pointer, for example:

```
std::tr1::bind(&my_proc, arg1, arg2 /* etc */); // works OK.
std::tr1::bind<double>(&my_proc, arg1, arg2 /* etc */); // causes compiler error.
std::tr1::bind<double>(my_function_object, arg1, arg2 /* etc */); // works OK.
```

On the other hand, the Boost implementation does work with pointers to overloaded functions, and optionally with function pointers with non-standard calling conventions.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Polymorphic function wrappers.

```
#include <boost/tr1/functional.hpp>
```

or

```
#include <functional>
```

The polymorphic function wrappers are a family of class templates that may be used as a generalized callback mechanism. A polymorphic function wrapper shares features with function pointers, in that both define a call interface (for example a function taking two integer arguments and returning a floating-point value) through which some arbitrary code may be called. However a polymorphic function wrapper can call any callable object with a compatible call signature, this could be a function pointer, or it could be a function object produced by std::tr1::bind, or some other mechanism. For more information see the Boost.Function documentation.

```
namespace std {
namespace tr1 {

// [3.7] polymorphic function wrappers
class bad_function_call;

template<class Function>
class function;

template<class Function>
void swap(function<Function>&, function<Function>&);

template<class Function1, class Function2>
void operator==(const function<Function1>&, const function<Function2>&);
template<class Function1, class Function2>
void operator!=(const function<Function1>&, const function<Function2>&);
template <class Function>
bool operator==(const function<Function>&, unspecified-null-pointer-type );
template <class Function>
bool operator==(unspecified-null-pointer-type , const function<Function>&);
template <class Function>
bool operator!=(const function<Function>&, unspecified-null-pointer-type );
template <class Function>
bool operator!=(unspecified-null-pointer-type , const function<Function>&);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_FUNCTION if your standard library implements this part of TR1.

Standard Conformity: The Boost version of `std::tr1::function` lacks the member function `target_type()` and does not inherit from `std::unary_function` or `std::binary_function` when applicable. The member function target() can only access pointer-to-member targets when they have been wrapped in mem_fn.

Type Traits.

```
#include <boost/tr1/type_traits.hpp>
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

or

```
#include <type_traits>
```

Type traits enable generic code to access the fundamental properties of a type, to determine the relationship between two types, or to transform one type into another related type. For more information refer to the Boost.Type_traits documentation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

template <class T, T v> struct integral_constant;

typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;

// [4.5.1] primary type categories:
template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;

// [4.5.2] composite type categories:
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;

// [4.5.3] type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_pod;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct has_trivial_constructor;
template <class T> struct has_trivial_copy;
template <class T> struct has_trivial_assign;
template <class T> struct has_trivial_destructor;
template <class T> struct has_nothrow_constructor;
template <class T> struct has_nothrow_copy;
template <class T> struct has_nothrow_assign;
template <class T> struct has_virtual_destructor;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;

// [4.6] type relations:
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;
template <class From, class To> struct is_convertible;

// [4.7.1] const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template <class T> struct add_cv;

// [4.7.2] reference modifications:
template <class T> struct remove_reference;
template <class T> struct add_reference;

// [4.7.3] array modifications:
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

// [4.7.4] pointer modifications:
template <class T> struct remove_pointer;
template <class T> struct add_pointer;

// [4.8] other transformations:
template <std::size_t Len, std::size_t Align> struct aligned_storage;

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_TYPE_TRAITS if your standard library implements this part of TR1.

Standard Conformity: No known problems.

Random Number Generators and Distributions.

```
#include <boost/tr1/random.hpp>
```

or

```
#include <random>
```

The random number library is devided into three parts: generators, which are nullary functors producing uniform random number distributions. Distributions, which are unary functors that adapt a generator to some specific kind of distribution. And the class template variate_generator which combines a generator with a distribution, to create a new generator. For more information see the Boost.Random documentation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

// [5.1.3] Class template variate_generator
template<class UniformRandomNumberGenerator, class Distribution>
class variate_generator;

// [5.1.4.1] Class template linear_congruential
template<class IntType, IntType a, IntType c, IntType m>
class linear_congruential;

// [5.1.4.2] Class template mersenne_twister
template<class UIntType, int w, int n, int m, int r,
UIntType a, int u, int s, UIntType b, int t, UIntType c, int l>
class mersenne_twister;

// [5.1.4.3] Class template substract_with_carry
template<class IntType, IntType m, int s, int r>
class subtract_with_carry;

// [5.1.4.4] Class template substract_with_carry_01
template<class RealType, int w, int s, int r>
class subtract_with_carry_01;

// [5.1.4.5] Class template discard_block
template<class UniformRandomNumberGenerator, int p, int r>
class discard_block;

// [5.1.4.6] Class template xor_combine
template<class UniformRandomNumberGenerator1, int s1,
class UniformRandomNumberGenerator2, int s2>
class xor_combine;

// [5.1.5] Predefined generators
typedef linear_congruential<
            implementation-defined ,
            16807,
            0,
            2147483647> minstd_rand0;

typedef linear_congruential<
            implementation-defined ,
            48271,
            0,
            2147483647> minstd_rand;

typedef mersenne_twister<
            implementation-defined ,
            32, 624, 397, 31,
            0x9908b0df, 11, 7,
            0x9d2c5680, 15,
            0xefc60000, 18> mt19937;

typedef subtract_with_carry_01<
            float,
            24,
            10,
            24> ranlux_base_01;

typedef subtract_with_carry_01<
            double,
            48,
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
            10,
            24> ranlux64_base_01;

typedef discard_block<
            subtract_with_carry<
                  implementation-defined ,
                  (1<<24),
                  10,
                  24>,
            223,
            24> ranlux3;

typedef discard_block<
            subtract_with_carry<
                  implementation-defined,
                  (1<<24),
                  10,
                  24>,
            389,
            24> ranlux4;

typedef discard_block<
            subtract_with_carry_01<
                  float,
                  24,
                  10,
                  24>,
            223,
            24> ranlux3_01;

typedef discard_block<
            subtract_with_carry_01<
                  float,
                  24,
                  10,
                  24>,
            389,
            24> ranlux4_01;

// [5.1.6] Class random_device
class random_device;

// [5.1.7.1] Class template uniform_int
template<class IntType = int>
class uniform_int;

// [5.1.7.2] Class bernoulli_distribution
class bernoulli_distribution;

// [5.1.7.3] Class template geometric_distribution
template<class IntType = int, class RealType = double>
class geometric_distribution;

// [5.1.7.4] Class template poisson_distribution
template<class IntType = int, class RealType = double>
class poisson_distribution;

// [5.1.7.5] Class template binomial_distribution
template<class IntType = int, class RealType = double>
class binomial_distribution;

// [5.1.7.6] Class template uniform_real
template<class RealType = double>
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
class uniform_real;

// [5.1.7.7] Class template exponential_distribution
template<class RealType = double>
class exponential_distribution;

// [5.1.7.8] Class template normal_distribution
template<class RealType = double>
class normal_distribution;

// [5.1.7.9] Class template gamma_distribution
template<class RealType = double>
class gamma_distribution;

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_RANDOM if your standard library implements this part of TR1.

Standard Conformity: The Boost implementation has the following limitations:

- The linear_congruential generator is fully supported for signed integer types only (unsigned types probably only work when the modulus is zero).
- The subtract_with_carry template does not support a modulus of zero.
- Not all of the standard generator types have Boost documentation yet, they are none the less supported however.
- Class template variate_generator does not have a template unary function call operator(), only the non-template nullary version.

Note also that most of the Random number generators have been re-implemented as thin wrappers around the Boost versions in order to provide a standard conforming interface (the Boost versions all take an additional, redundant, template parameter, and are initialized by iterators rather than functors).

Tuples.

```
#include <boost/tr1/tuple.hpp>
```

or

```
#include <tuple>
```

A tuple is a fixed size collection of elements. Pairs, triples, quadruples etc. are tuples. In a programming language, a tuple is a data object containing other objects as elements. These element objects may be of different types. Tuples are convenient in many circumstances. For instance, tuples make it easy to define functions that return more than one value. Some programming languages, such as ML, Python and Haskell, have built-in tuple constructs. Unfortunately C++ does not. To compensate for this "deficiency", the TR1 Tuple Library implements a tuple construct using templates. For more information see the Boost Tuple Library Documentation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

// [6.1.3] Class template tuple
template <class T1 = unspecified ,
class T2 = unspecified ,
...,
class TM = unspecified > class tuple;

// [6.1.3.2] Tuple creation functions
const unspecified ignore;

template<class T1, class T2, ..., class TN>
tuple<V1, V2, ..., VN> make_tuple(const T1&, const T2& , ..., const TN&);

// [6.1] Tuple types Containers
template<class T1, class T2, ..., class TN>
tuple<T1&, T2&, ..., TN&> tie(T1&, T2& , ..., TN&);

// [6.1.3.3] Tuple helper classes
template <class T> class tuple_size;
template <int I, class T> class tuple_element;

// [6.1.3.4] Element access
template <int I, class T1, class T2, ..., class TN>
RI get(tuple<T1, T2, ..., TN>&);
template <int I, class T1, class T2, ..., class TN>
PI get(const tuple<T1, T2, ..., TN>&);

// [6.1.3.5] relational operators
template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator==(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);
template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator<(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);
template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator!=(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);
template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator>(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);
template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator<=(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);
template<class T1, class T2, ..., class TM, class U1, class U2, ..., class UM>
bool operator>=(const tuple<T1, T2, ..., TM>&, const tuple<U1, U2, ..., UM>&);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_TUPLE if your standard library implements this part of TR1.

Standard Conformity: No known issues for conforming compilers.

Tuple Interface to std::pair.

```
#include <boost/tr1/utility.hpp>
```

or

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
#include <utility>
```

The existing class template std::pair, can also be accessed using the tuple interface.

```
namespace std {
namespace tr1 {

template <class T> class tuple_size; // forward declaration
template <int I, class T> class tuple_element; // forward declaration
template <class T1, class T2> struct tuple_size<std::pair<T1, T2> >;
template <class T1, class T2> struct tuple_element<0, std::pair<T2, T2> >;
template <class T1, class T2> struct tuple_element<1, std::pair<T2, T2> >;
// see below for definition of "P".
template<int I, class T1, class T2> P& get(std::pair<T1, T2>&);
template<int I, class T1, class T2> const P& get(const std::pair<T1, T2>&);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_UTILITY if your standard library implements this part of TR1.

Standard Conformity: No known problems.

Fixed Size Array.

```
#include <boost/tr1/array.hpp>
```

or

```
#include <array>
```

Class template array is a fixed size array that is safer than and no less efficient than a C style array. Class array fulfils almost all of the requirements of a reversible-container (see Section 23.1, [lib.container.requirements] of the C++ Standard). For more information refer to the Boost.Array documentation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

// [6.2.2] Class template array
template <class T, size_t N > struct array;

// Array comparisons
template <class T, size_t N> bool operator== (const array<T,N>& x, const array<T,N>& y);
template <class T, size_t N> bool operator< (const array<T,N>& x, const array<T,N>& y);
template <class T, size_t N> bool operator!= (const array<T,N>& x, const array<T,N>& y);
template <class T, size_t N> bool operator> (const array<T,N>& x, const array<T,N>& y);
template <class T, size_t N> bool operator>= (const array<T,N>& x, const array<T,N>& y);
template <class T, size_t N> bool operator<= (const array<T,N>& x, const array<T,N>& y);

// [6.2.2.2] Specialized algorithms
template <class T, size_t N > void swap(array<T,N>& x, array<T,N>& y);

// [6.2.2.5] Tuple interface to class template array
template <class T> class tuple_size; // forward declaration
template <int I, class T> class tuple_element; // forward declaration
template <class T, size_t N> struct tuple_size<array<T, N> >;
template <int I, class T, size_t N> struct tuple_element<I, array<T, N> >;
template <int I, class T, size_t N> T& get( array<T, N>&);
template <int I, class T, size_t N> const T& get(const array<T, N>&);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_ARRAY if your standard library implements this part of TR1.

Standard Conformity: No known issues as of Boost-1.34 onwards.

Hash Function Objects.

```
#include <boost/tr1/functional.hpp>
```

or

```
#include <functional>
```

Class template std::hash is a unary-functor that converts some type T into a hash-value, specializations of std::hash are provided for integer, character, floating point, and pointer types, plus the two string types std::string and std::wstring. See the Boost.Hash documentation for more information.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

template <class T>
struct hash : public unary_function<T, size_t>
{
   size_t operator()(T val)const;
};

// Hash function specializations
template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<wchar_t>;
template <> struct hash<short>;
template <> struct hash<int>;
template <> struct hash<long>;
template <> struct hash<unsigned short>;
template <> struct hash<unsigned int>;
template <> struct hash<unsigned long>;
template <> struct hash<float>;
template <> struct hash<double>;
template <> struct hash<long double>;
template<class T> struct hash<T*>;
template <> struct hash<std::string>;
template <> struct hash<std::wstring>;

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_HASH if your standard library implements this part of TR1.

Standard Conformity: Boost.Hash adds specialisations of std::hash for a wider range of types than those required by TR1: Boost.Hash acts as a testbed for issue 6.18 in the Library Extension Technical Report Issues List.

Regular Expressions.

```
#include <boost/tr1/regex.hpp>
```

or

```
#include <regex>
```

This library provides comprehensive support for regular expressions, including either iterator or string based matching, searching, search-and-replace, iteration, and tokenization. Both POSIX and ECMAScript (JavaScript) regular expressions are supported. For more information see the Boost.Regex documentation.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

// [7.5] Regex constants
namespace regex_constants {

typedef bitmask_type syntax_option_type;
typedef bitmask_type match_flag_type;
typedef implementation-defined error_type;

} // namespace regex_constants

// [7.6] Class regex_error
class regex_error;

// [7.7] Class template regex_traits
template <class charT> struct regex_traits;

// [7.8] Class template basic_regex
template <class charT, class traits = regex_traits<charT> >
class basic_regex;

typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

// [7.8.6] basic_regex swap
template <class charT, class traits>
void swap(basic_regex<charT, traits>& e1,
          basic_regex<charT, traits>& e2);

// [7.9] Class template sub_match
template <class BidirectionalIterator>
class sub_match;

typedef sub_match<const char*> csub_match;
typedef sub_match<const wchar_t*> wcsub_match;
typedef sub_match<string::const_iterator> ssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;

// [7.9.2] sub_match non-member operators

/* Comparison operators omitted for clarity.... */

template <class charT, class ST, class BiIter>
basic_ostream<charT, ST>&
   operator<<(basic_ostream<charT, ST>& os,
              const sub_match<BiIter>& m);

// [7.10] Class template match_results
template <class BidirectionalIterator,
          class Allocator = allocator<sub_match<BidirectionalIterator> > >
class match_results;

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;

// match_results comparisons
template <class BidirectionalIterator, class Allocator>
bool operator== (const match_results<BidirectionalIterator, Allocator>& m1,
                 const match_results<BidirectionalIterator, Allocator>& m2);
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template <class BidirectionalIterator, class Allocator>
bool operator!= (const match_results<BidirectionalIterator, Allocator>& m1,
                 const match_results<BidirectionalIterator, Allocator>& m2);

// [7.10.6] match_results swap
template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);

// [7.11.2] Function template regex_match
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_match(BidirectionalIterator first,
                 BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);

template <class BidirectionalIterator, class charT, class traits>
bool regex_match(BidirectionalIterator first,
                 BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);

template <class charT, class Allocator, class traits>
bool regex_match(const charT* str,
                 match_results<const charT*, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator,Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);

template <class charT, class traits>
bool regex_match(const charT* str,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);

template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags = regex_constants::match_default);

// [7.11.3] Function template regex_search
template <class BidirectionalIterator, class Allocator, class charT, class traits>
bool regex_search(BidirectionalIterator first,
                  BidirectionalIterator last,
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

template <class BidirectionalIterator, class charT, class traits>
bool regex_search(BidirectionalIterator first,
                  BidirectionalIterator last,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

template <class charT, class Allocator, class traits>
bool regex_search(const charT* str,
                  match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
                  regex_constants::match_flag_type flags = regex_constants::match_default);

template <class charT, class traits>
bool regex_search(const charT* str,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

template <class ST, class SA, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

template <class ST, class SA, class Allocator, class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags = regex_constants::match_default);

// [7.11.4] Function template regex_replace
template <class OutputIterator, class BidirectionalIterator, class traits, class charT>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const basic_regex<charT, traits>& e,
                             const basic_string<charT>& fmt,
                             regex_constants::match_flag_type flags = regex_constants::match_default);

template <class traits, class charT>
basic_string<charT> regex_replace(const basic_string<charT>& s,
                                  const basic_regex<charT, traits>& e,
                                  const basic_string<charT>& fmt,
                                  regex_constants::match_flag_type flags = regex_constants::match_d

// [7.12.1] Class template regex_iterator
template <class BidirectionalIterator,
          class charT = typename iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
class regex_iterator;

typedef regex_iterator<const char*> cregex_iterator;
typedef regex_iterator<const wchar_t*> wcregex_iterator;
typedef regex_iterator<string::const_iterator> sregex_iterator;
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;

// [7.12.2] Class template regex_token_iterator
template <class BidirectionalIterator,
          class charT = typename iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
class regex_token_iterator;

typedef regex_token_iterator<const char*> cregex_token_iterator;
typedef regex_token_iterator<const wchar_t*> wcregex_token_iterator;
typedef regex_token_iterator<string::const_iterator> sregex_token_iterator;
typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_REGEX if your standard library implements this part of TR1.

Standard Conformity: No known problems.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Complex Number Algorithm Overloads.

```
#include <boost/tr1/complex.hpp>
```

or

```
#include <complex>
```

The following function templates have additional overloads: `arg`, `norm`, `conj`, `polar`, `imag`, and `real`.

The additional overloads are sufficient to ensure:

- If the argument has type `long double`, then the overload behaves as if the argument had been cast to `std::complex<long double>`.
- Otherwise, if the argument has type `double` or is an integer type, then the overload behaves as if the argument had been cast to `std::complex<double>`.
- Otherwise, if the argument has type `float`, then the overload behaves as if the argument had been cast to `std::complex<float>`.

The function template `pow` has additional overloads sufficient to ensure, for a call with at least one argument of type `std::complex<T>`:

- If either argument has type `complex<long double>` or type `long double`, then the overload behaves as if both arguments were cast to `std::complex<long double>`
- Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then the overload behaves as if both arguments were cast to `std::complex<double>`
- Otherwise, if either argument has type `complex<float>` or `float`, then the overload behaves as if both arguments were cast to `std::complex<float>`

In the following synopsis, `Real` is a floating point type, `Arithmetic` is an integer or floating point type, and `PROMOTE(X1 ... XN)` is the largest floating point type in the list X1 to XN, after any non-floating point types in the list have been replaced by the type `double`.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
template <class Arithmetic>
PROMOTE(Arithmetic) arg(const Arithmetic& t);

template <class Arithmetic>
PROMOTE(Arithmetic) norm(const Arithmetic& t);

template <class Arithmetic>
complex<PROMOTE(Arithmetic)> conj(const Arithmetic& t);

template <class Arithmetic1, class Arithmetic2>
complex<PROMOTE(Arithmetic1,Arithmetic2)> polar(const Arithmetic1& rho, const Arithmetic2& theta = 0);

template <class Arithmetic>
PROMOTE(Arithmetic) imag(const Arithmetic& );

template <class Arithmetic>
PROMOTE(Arithmetic) real(const Arithmetic& t);

template<class Real1, class Real2>
complex<PROMOTE(Real1, Real2)>
   pow(const complex<Real1>& x, const complex<Real2>& y);

template<class Real, class Arithmetic>
complex<PROMOTE(Real, Arithmetic)>
   pow (const complex<Real>& x, const Arithmetic& y);

template<class Arithmetic, class Real>
complex<PROMOTE(Real, Arithmetic)>
   pow (const Arithmetic& x, const complex<Real>& y);
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_COMPLEX_OVERLOADS if your standard library implements the additional overloads for the existing complex arithmetic functions.

Standard Conformity: No known problems.

Complex Number Additional Algorithms.

```
#include <boost/tr1/complex.hpp>
```

or

```
#include <complex>
```

The algorithms `acos`, `asin`, `atan`, `acosh`, `asinh`, `atanh` and `fabs` are overloaded for arguments of type `std::complex<T>`. These algorithms are entirely classical, and behave as specified in the C99 standard section 7.3.5. See the Boost.Math documentation for more information.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

template<class T> complex<T> acos(complex<T>& x);
template<class T> complex<T> asin(complex<T>& x);
template<class T> complex<T> atan(complex<T>& x);
template<class T> complex<T> acosh(complex<T>& x);
template<class T> complex<T> asinh(complex<T>& x);
template<class T> complex<T> atanh(complex<T>& x);
template<class T> complex<T> fabs(complex<T>& x);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_COMPLEX_INVERSE_TRIG if your standard library implements the additional inverse trig functions.

Standard Conformity: No known problems.

TR By Subject: Unsupported Features

Mathematical Special Functions.

The TR adds 23 special functions (plus float and long double overloads) to header <cmath>. However, at present there is no Boost License compatible implementation of these functions, so these are **unsupported by this implementation** unless your standard library supports them itself.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

// [5.2.1.1] associated Laguerre polynomials:
double assoc_laguerre(unsigned n, unsigned m, double x);
float assoc_laguerref(unsigned n, unsigned m, float x);
long double assoc_laguerrel(unsigned n, unsigned m, long double x);

// [5.2.1.2] associated Legendre functions:
double assoc_legendre(unsigned l, unsigned m, double x);
float assoc_legendref(unsigned l, unsigned m, float x);
long double assoc_legendrel(unsigned l, unsigned m, long double x);

// [5.2.1.3] beta function:
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);

// [5.2.1.4] (complete) elliptic integral of the first kind:
double comp_ellint_1(double k);
float comp_ellint_1f(float k);
long double comp_ellint_1l(long double k);

// [5.2.1.5] (complete) elliptic integral of the second kind:
double comp_ellint_2(double k);
float comp_ellint_2f(float k);
long double comp_ellint_2l(long double k);

// [5.2.1.6] (complete) elliptic integral of the third kind:
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);

// [5.2.1.7] confluent hypergeometric functions:
double conf_hyperg(double a, double c, double x);
float conf_hypergf(float a, float c, float x);
long double conf_hypergl(long double a, long double c, long double x);

// [5.2.1.8] regular modified cylindrical Bessel functions:
double cyl_bessel_i(double nu, double x);
float cyl_bessel_if(float nu, float x);
long double cyl_bessel_il(long double nu, long double x);

// [5.2.1.9] cylindrical Bessel functions (of the first kind):
double cyl_bessel_j(double nu, double x);
float cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);

// [5.2.1.10] irregular modified cylindrical Bessel functions:
double cyl_bessel_k(double nu, double x);
float cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);

// [5.2.1.11] cylindrical Neumann functions;
// cylindrical Bessel functions (of the second kind):
double cyl_neumann(double nu, double x);
float cyl_neumannf(float nu, float x);
long double cyl_neumannl(long double nu, long double x);

// [5.2.1.12] (incomplete) elliptic integral of the first kind:
double ellint_1(double k, double phi);
float ellint_1f(float k, float phi);
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
long double ellint_1l(long double k, long double phi);

// [5.2.1.13] (incomplete) elliptic integral of the second kind:
double ellint_2(double k, double phi);
float ellint_2f(float k, float phi);
long double ellint_2l(long double k, long double phi);

// [5.2.1.14] (incomplete) elliptic integral of the third kind:
double ellint_3(double k, double nu, double phi);
float ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);

// [5.2.1.15] exponential integral:
double expint(double x);
float expintf(float x);
long double expintl(long double x);

// [5.2.1.16] Hermite polynomials:
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);

// [5.2.1.17] hypergeometric functions:
double hyperg(double a, double b, double c, double x);
float hypergf(float a, float b, float c, float x);
long double hypergl(long double a, long double b, long double c, long double x);

// [5.2.1.18] Laguerre polynomials:
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

// [5.2.1.19] Legendre polynomials:
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);

// [5.2.1.20] Riemann zeta function:
double riemann_zeta(double);
float riemann_zetaf(float);
long double riemann_zetal(long double);

// [5.2.1.21] spherical Bessel functions (of the first kind):
double sph_bessel(unsigned n, double x);
float sph_besself(unsigned n, float x);
long double sph_bessell(unsigned n, long double x);

// [5.2.1.22] spherical associated Legendre functions:
double sph_legendre(unsigned l, unsigned m, double theta);
float sph_legendref(unsigned l, unsigned m, float theta);
long double sph_legendrel(unsigned l, unsigned m, long double theta);

// [5.2.1.23] spherical Neumann functions;
// spherical Bessel functions (of the second kind):
double sph_neumann(unsigned n, double x);
float sph_neumannf(unsigned n, float x);
long double sph_neumannl(unsigned n, long double x);

} // namespace tr1
} // namespace std
```

Standard Conformity: ***Not Supported.***

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Unordered Associative Set (Hash Table).

```
#include <boost/tr1/unordered_set.hpp>
```

or

```
#include <unordered_set>
```

This is not currently supported by Boost, although that situation is hoped to change soon.

```
namespace std {
namespace tr1 {

template <class Value,
         class Hash = hash<Value>,
         class Pred = std::equal_to<Value>,
         class Alloc = std::allocator<Value> >
class unordered_set;

// [6.3.4.5] Class template unordered_multiset
template <class Value,
         class Hash = hash<Value>,
         class Pred = std::equal_to<Value>,
         class Alloc = std::allocator<Value> >
class unordered_multiset;

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
         unordered_set<Value, Hash, Pred, Alloc>& y);

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
         unordered_multiset<Value, Hash, Pred, Alloc>& y);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_UNORDERED_SET if your standard library implements this part of TR1.

Standard Conformity: Not supported.

Unordered Associative Map (Hash Table).

```
#include <boost/tr1/unordered_map.hpp>
```

or

```
#include <unordered_map>
```

This is not currently supported by Boost, although that situation is hoped to change soon.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
namespace std {
namespace tr1 {

// [6.3.4.4] Class template unordered_map
template <class Key,
          class T,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_map;

// [6.3.4.6] Class template unordered_multimap
template <class Key,
          class T,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_multimap;

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_map<Key, T, Hash, Pred, Alloc>& y);

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
          unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

} // namespace tr1
} // namespace std
```

Configuration: Boost.Config should (automatically) define the macro BOOST_HAS_TR1_UNORDERED_MAP if your standard library implements this part of TR1.

Standard Conformity: Not supported.

TR1 By Header

<array>

See: Fixed Size Array

<cmath>

See: Special Functions

<complex>

See: Additional Overloads for Complex Number Algorithms

See: Additional Complex Number Algorithms

<functional>

See: Reference Wrapper.

See: Result_of.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

See: Member Function Wrappers.

See: Function Binders.

See: Polymorphic Function Wrappers.

See: Hash Functions.

<memory>

See: Smart Pointers.

<random>

See: Random Numbers.

<regex>

See: Regular Expressions.

<tuple>

See: Tuple Types.

<type_traits>

See: Type Traits.

<unordered_map>

See: Unordered Associative Map

<unordered_set>

See: Unordered Associative Set.

<utility>

See: Tuple Interface to std::pair.

Implementation

When Boost.TR1 is configured to make use of your standard library's native TR1 implementation, then it doesn't do very much: it just includes the appropriate header.

When Boost.TR1 is using the Boost implementation of a particular component, then it includes the appropriate Boost header(s) and imports the necessary declarations in `namespace std::tr1` with using declarations. Note that only those declarations that are part of the standard are imported: the implementation is deliberately quite strict about not including any Boost-specific extensions in `namespace std::tr1`, in order to catch any portability errors in user code. If you really need to use Boost-specific extensions then you should include the Boost headers directly and use the declarations in `namespace boost::` instead. Note that this style of implementation is not completely standards-conforming, in particular it is not possible to add user-defined template specializations of TR1 components into `namespace std::tr1`. There are also one or two Boost libraries that are not yet fully standards conforming, any such non-conformities are documented in the TR1 by subject section. Hopefully, occurrences of non-standard behavior should be extremely rare in practice however.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

If you use the standard conforming header includes (in `boost/tr1/tr1`) then these header names can sometimes conflict with existing standard library headers (for example `shared_ptr` is added to the existing standard library header `<memory>` rather than it's own header). These headers forward on to your existing standard library header in one of two ways: for gcc it uses `#include_next`, and for other compilers it uses the macro `BOOST_TR1_STD_HEADER(header)` (defined in boost/tr1/detail/config.hpp) which evaluates to `#include <../include/header>`. This should work "straight out the box" for most compilers, but does mean that these headers should **never** be placed inside a directory called "include" that is already in your compiler's search path.

Testing

The test suite for Boost.TR1 is relatively lightweight; tests have been added to the Boost.Config test suite for each new configuration macro, and each TR1 component has a very short concept check test added. The concept test programs are designed only to verify that all the TR1 components that are supposed to be in `namespace std::tr1` are indeed present and have standards conforming interfaces. There are a few test programs (those which end in the suffix "_tricky") which do not currently compile with the Boost.TR1 implementation, because the relevant Boost libraries have not yet implemented the features tested; hopefully these incompatibilities will be removed in future releases.

The concept tests do not take account of compiler defects (quite deliberately so); the intent is that the tests can be used to verify conformance with the standard, both for Boost code, and for third party implementations. Consequently very many of these tests are known to fail with older compilers. This should not be taken as evidence that these compilers can not be used at all with Boost.TR1, simply that there are features missing that make those compilers non-conforming.

Full runtime tests for TR1 components are not in general part of this test suite, however, it is hoped that the Boost.TR1 component authors will make their regular test suites compile with the standards conforming headers as well as the Boost-specific ones. This will allow these tests to be used against the standard library's own TR1 implementation as well as the Boost one.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/